(19) BUNDESREPUBLIK DEUTSCHLAND

DEUTSCHES PATENT- UND MARKENAMT

(12) **Patentschrift**

(10) **DE 199 26 467 C 1**

(51) Int. Cl.[7]: G 06 F 11/36

(21) Aktenzeichen: 199 26 467.8-53
(22) Anmeldetag: 10. 6. 1999
(43) Offenlegungstag: –
(45) Veröffentlichungstag der Patenterteilung: 11. 1. 2001

Innerhalb von 3 Monaten nach Veröffentlichung der Erteilung kann Einspruch erhoben werden

(73) Patentinhaber:
Siemens AG, 80333 München, DE

(72) Erfinder:
Stroetmann, Karl, Dr., 81739 München, DE

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:
US 56 68 999 A

(54) Verfahren zum Betreiben eines Computersystems, Bytecode-Verifier und Computersystem

(57) Die Erfindung betrifft ein Verfahren zum Betreiben eines Computersystems sowie einen Bytecode-Verifier und ein Computersystem.
Mit dem erfindungsgemäßen Verfahren wird geprüft, ob ein auf ein Computersystem geladenes Computerprogramm einen unzulässigen Zugriff auf eine Variable ausübt, das heißt, ob die Variable, bevor sie gelesen wird, initialisiert ist. Diese Prüfung erfolgt vor der Ausführung des Programms, so daß beim Ausführen des Programms eine derartige Prüfung nicht mehr durchgeführt werden muß.
Das erfindungsgemäße Verfahren benötigt bei der Prüfung des Programms wenig Speicherplatz und führt dennoch eine vollständige Prüfung durch. Da die Prüfung vor der Ausführung des Programms erfolgt, wird die Programmausführung selbst erheblich beschleunigt, da hierbei keine weitere Prüfung notwendig ist.

DE 199 26 467 C 1

DE 199 26 467 C 1

NSDOCID: <DE_____19926467C1_I_>

Beschreibung

Die Erfindung betrifft ein Verfahren zum Betreiben eines Computersystems, einen Bytecode-Verifier und ein Computersystem.

Aus der US 5,668,999 geht ein Verfahren zum Betreiben eines Computersystems hervor, das nach dem Laden eines Computerprogrammes und vor dem Ausführen des Computerprogrammes dieses überprüft, ob es keine unzulässigen Datenverarbeitungsvorgänge ausführt. Hierzu ist ein sogenannter Bytecode-Verifier vorgesehen, der den Bytecode des geladenen Computerprogrammes überprüft. Der Verifier enthält einen virtuellen Stack zum vorübergehenden Speichern von Typinformationen, die die Daten spezifizieren, die bei der Ausführung des Programmes auf dem Stack liegen. Der Verifier umfaßt ferner einen Stack-Schnappschußspeicher mit einem Schnappschußverzeichnis und einem Schnappschußspeicherbereich zum Speichern des Zustandes des virtuellen Stacks an unterschiedlichen Punkten des Programmablaufes.

Die Überprüfung des Computerprogrammes erfolgt in zwei Phasen, wobei in der ersten Phase alle Zieladressen ermittelt werden, auf die bedingte oder unbedingte Sprünge zeigen. In der zweiten Phase wird das Programm simuliert. Dabei werden sogenannte Schnappschüsse des virtuellen Stacks zu jeder Zieladresse des Sprungs erstellt und in das Schnappschußverzeichnis abgespeichert. Durch Vergleich der im Schnappschußverzeichnis gespeicherten Informationen wird festgestellt, ob der Stack in einer Art und Weise manipuliert wird, die zu einem Overflow oder Underflow führen könnte. Sollte eine derartige unzulässige Änderung verursacht werden, wird eine Fehlermeldung ausgegeben.

Die Verwendung eines derartigen Bytecode-Verifiers erlaubt eine erhebliche Beschleunigung der Ausführung des Programms. Herkömmliche Interpreter zur Ausführung eines Computerprogrammes enthalten Routinen zum Überwachen des Stacks und zur Vermeidung eines Overflows bzw. Underflows. Diese Routinen wurden bei jedem auf den Stack zugreifenden Befehl ausgeführt. Das Überwachen des Stacks konnte deshalb bis zu 80% der Ausführungszeit eines Computerprogrammes in Anspruch nehmen. Die Verwendung des Bytecode-Verifiers ist besonders bei Schleifen von Vorteil, da lediglich die Befehle eines Schleifendurchgangs ein einziges Mal mit dem Bytecode-Verifier überprüft werden, wohingegen bei einem herkömmlichen Interpreter die einzelnen Befehle in jedem Schleifendurchgang erneut überprüft worden sind.

Der virtuelle Stack ist wie ein regulärer Stack aufgebaut, außer daß darin anstelle der tatsächlichen Daten und Konstanten Indikatoren gespeichert werden, die den jeweiligen Datentyp, wie z. B. Ganzzahl (I), lange Ganzzahl (L), einfach-genaue Fließkommazahl (F), doppelt-genaue Fließkommazahl (D), Byte (B), usw. darstellen.

Bei diesem bekannten Verfahren wird auch ein virtuelles Feld der lokalen Variablen angelegt. Dieses virtuelle Feld ist ähnlich wie der virtuelle Stack aufgebaut, das heißt, daß in dem virtuellen Feld lediglich die Indikatoren der Datentypen der in den tatsächlichen Speicherfeld abzuspeichernden Daten enthalten sind.

Bei Sprüngen im Programmablauf werden jedoch lediglich Schnappschüsse des virtuellen Stacks miteinander verglichen. Änderungen im virtuellen Feld bleiben unberücksichtigt, so daß aufgrund von Sprüngen erzeugte Fehler beim Zugriff auf das Feld der lokalen Variablen nicht festgestellt werden können. Dies kann zu Fehlern im Programmablauf führen, die den Betrieb des Computersystems beeinträchtigen und sogar zu einem Absturz des Computersystems führen können.

In der US 5,668,999 ist der Bytecode-Verifier anhand der Computersprache OAK beschrieben. Aus OAK ist die Computersprache Java™ entwickelt worden. Java™ wird wegen ihrer Plattformunabhängigkeit oft bei Internetanwendungen verwendet, die von einem Server auf einen beliebigen Client geladen werden, wobei der Client ein Computersystem mit einem beliebigen, gebräuchlichen Betriebssystem sein kann.

In der Java™ Virtual Machine Specification, Tim Lindholm und Frank Yellin, Addison-Wesley, 1997 (ISBN 0-201-63452-X), S. 128 ff ist ein Bytecode-Verifier beschrieben, der im wesentlichen demjenigen aus der US 5,668,999 entspricht, sich aber von diesem dadurch unterschiedet, daß mit den Schnappschüssen nicht nur Kopien des virtuellen Stacks, sondern auch Kopien des virtuellen Feldes der lokalen Variablen erstellt werden. Bei einem bedingten oder unbedingten Sprung im Programmablauf werden nicht nur die Schnappschüsse der Stacks sondern auch die der virtuellen Felder der lokalen Variablen verglichen, so daß auch ein durch einen nach einem Sprung ausgeführten Zugriff auf das Variablenfeld erzeugter Fehler festgestellt werden kann.

Das Erstellen einer vollständigen Kopie des virtuellen Feldes der lokalen Variablen für die Zieladresse bei einem Sprung erzeugt eine erhebliche Datenmenge, die dementsprechend viel Speicherplatz verbraucht. Da OAK und Java nicht nur für Internetanwendungen, sondern insbesondere für embedded-Anwendungen mit kleinen Prozessorsystemen entwickelt worden sind, steht dieser erhebliche Speicherbedarf der ursprünglichen Zielsetzung entgegen. Insbesondere für Spezialentwicklungen von Java, wie z. B. JAVACARD, das eine speziell für Chipkarten entwickelte Programmiersprache ist, besteht ein erheblicher Bedarf nach einer Verringerung des Speicherbedarfs.

Der Erfindung liegt deshalb die Aufgabe zugrunde, ein Verfahren der in der US 5,668,999 beschriebenen Art und Weise, einen Bytecode-Verifier und ein Computersystem derart weiterzuentwickeln, daß bei geringem Speicherbedarf dennoch eine sichere Ausführung eines auf einem Computersystem geladenem Computerprogrammes gewährleistet wird.

Die Aufgabe wird durch ein Verfahren mit den Merkmalen des Anspruchs 1, durch einen Bytecode-Verifier mit den Merkmalen des Anspruchs 9 und durch ein Computersystem mit den Merkmalen des Anspruchs 12 gelöst. Vorteilhafte Ausgestaltungen der Erfindung sind in den Unteransprüchen angegeben.

Das erfindungsgemäße Verfahren zum Betreiben eines Computersystems umfaßt folgende Schritte:

a) Speichern eines Computerprogramms in einem Speicher des Computersystems, wobei das Computerprogramm eine Folge von Bytecodes umfaßt, die Programmbefehle darstellen, wobei die Programmbefehle in der Reihenfolge eines vorbestimmten Programmablaufes bei der Ausführung des Computerprogrammes abgearbeitet werden und auf Variablen zugreifen können,

b) Prüfen des in das Computersystem geladenen Computerprogrammes vor dem Ausführen des Computerpro-

gramms, ob die darin enthaltenen Programmbefehle beim Ausführen einen unzulässigen Datenverarbeitungsvorgang erzeugen würden, und

c) falls das Prüfen des Computerprogramms keinen unzulässigen Datenverarbeitungsvorgang ergeben hat, wird die Ausführung des Programms erlaubt, ansonsten wird eine entsprechende Fehlermeldung ausgegeben.

Das erfindungsgemäße Verfahren zeichnet sich dadurch aus, daß beim Prüfen des Computerprogramms die einzelnen Programmbefehle in der Reihenfolge des Programmablaufes überprüft werden, ob sie bei Ihrer Ausführung auf eine Variable zugreifen, bevor diese initialisiert worden ist.

Der Erfinder hat erkannt, daß mit dem aus der US 5,668,999 bekannten Verfahren bei Sprüngen im Programmablauf unzulässige Datenverarbeitungsvorgänge nicht erkannt werden. In **Fig.** 3 ist ein Programmbeispiel gezeigt, das einen unzulässigen Datenverarbeitungsvorgang aufweist, der von diesem bekannten Verfahren nicht erkannt wird. Bei der Darstellung in **Fig.** 3 ist in der linken Spalte der jeweilige Programmbefehl, in der mittleren Spalte der Inhalt des virtuellen Stacks und in der rechten Spalte der Inhalt des virtuellen Feldes der lokalen Variable dargestellt. Die Programmbefehle sind aufeinanderfolgend numeriert. Mit dem ersten Befehl "goto 5" springt der Programmablauf auf den Befehl Nummer 5 (aload 2). Hierbei wird der Inhalt der Variable 2 des virtuellen Feldes auf den Stack geladen. Die Variable 2 sollte einen Indikator für einer Datentyp, wie z. B. "C" beinhalten. Da jedoch die Programmbefehle 2 bis 4 übersprungen worden sind und mit dem Befehl 3 (astore 2) die Variable 2 zum erstenmal mit einem Wert belegt werden würde, ist die Variable 2 bei der Ausführung des Befehles 5 noch nicht mit einem Wert belegt. Das heißt, daß die Variable 2 bei der Ausführung des Befehles 5 noch nicht initialisiert ist. Würde das Programm ausgeführt werden, würde mit dem Befehl 5 die Variable 2 gelesen werden, obwohl hier noch kein Wert zugewiesen worden ist. Dies ergibt einen undefinierten Zustand, da in unzulässiger Weise nicht-vorhandene Daten gelesen werden. Derartige Programmausführungen können zu undefinierten Zuständen des Computersystems führen oder sogar einen Absturz des gesamten Computersystems hervorrufen.

Mit dem erfindungsgemäßen Verfahren wird geprüft, ob beim Zugriff auf eine Variable (Befehl Nr. 5) die Variable vorher initialisiert worden ist. Bei der Prüfung werden die einzelnen Befehle des Computerprogrammes in der Reihenfolge untersucht, mit der sie bei einer Ausführung des Computerprogrammes ausgeführt werden, wobei bei einer Verzweigung des Programmablaufes die einzelnen Programmzweige unabhängig voneinander untersucht werden. Nur wenn sichergestellt ist, daß die Variablen vor ihrem Zugriff initialisiert sind, das heißt mit einem Wert belegt sind, wird die Ausführung des Programms erlaubt. Hierdurch werden ein unzulässiger Zugriff auf eine Variable sicher verhindert und undefinierte Zustände des Computersystems vermieden.

Der Bedarf an zusätzlichem Speicherplatz eines Bytecode-Verifiers, der mit einer Routine zum Prüfen des Computerprogrammes, ob vor den Zugriffen auf Variablen diese vorher initialisiert worden sind, versehen ist, ist wesentlich geringer, als wenn ein virtuelles Feld lokaler Variablen für jede Zielstelle eines bedingten oder unbedingten Sprunges kopiert werden würde.

Mit dem erfindungsgemäßen Verfahren wird somit bei geringem Speicherplatzbedarf eine sichere Ausführung eines auf einem Computersystem geladenem Computerprogrammes gewährleistet, da bei der Ermittlung von unzulässigen Datenverarbeitungsvorgängen auch unzulässige Datenzugriffe im Feld für lokale Variablen festgestellt werden.

Die Erfindung wird nachfolgend beispielhaft näher anhand der Zeichnung beschrieben, in denen zeigen:

Fig. 1 schematisch vereinfacht ein Computersystem, an welchem das erfindungsgemäße Verfahren ausgeführt werden kann,

Fig. 2 das erfindungsgemäße Verfahren in einem Flußdiagramm,

Fig. 3 ein Programmbeispiel,

Fig. 4 ein Ablaufschema des Programmbeispiels aus **Fig.** 3, und

Fig. 5 ein weiteres Ablaufschema mit einer Verzweigung des Programmablaufes.

In **Fig.** 1 ist ein Computersystem gezeigt, an dem das erfindungsgemäße Verfahren ausgeführt werden kann. Das Computersystem **1** weist eine zentrale Recheneinheit (CPU) **2** auf, die in an sich bekannter Weise mit einem flüchtigen Speicher (RAM) **3** und einem nichtflüchtigen Speicher **4** zusammenwirkt. Der nichtflüchtige Speicher **4** besteht in der Regel aus einer Festplatte, auf welcher die zum Betreiben des Computersystems **1** notwendige Programme, wie z. B. das Betriebssystem, und weitere Anwendungsprogramme gespeichert sind. Das Computersystem **1** weist eine I/O-Einheit **5** auf, mit welcher ein Benutzer Daten in das Computersystem **1** eingeben kann bzw. an welcher dem Benutzer Daten angezeigt werden. Eine solche I/O-Einheit **5** besteht in der Regel aus einer Tastatur, einem Bildschirm und einem Diskettenlaufwerk. Mittels eines Modems **6** kann eine Datenverbindung zu einem weiteren Computersystem hergestellt werden. Von diesem weiteren Computersystem können beispielsweise Programme auf das vorliegende Computersystem **1** geladen und im Speicher **3**, **4** abgespeichert werden.

Zur Überprüfung der auf das Computersystem **1** geladenen Computerprogramme ist ein Bytecode-Verifier an dem Computersystem **1** installiert. Der Bytecode-Verifier entspricht im wesentlichen dem aus der US 5,668,999 oder aus Java™ Virtual Machine Specification, Tim Lindholm und Frank Yellin, Addison-Wesley, 1997 (ISBN 0-201-63452-X), Seiten 128 ff bekannten Bytecode-Verifier. Derartige Bytecode-Verifier werden insbesondere bei plattformunabhängigen Computersprachen wie OAK und Java eingesetzt.

Erfindungsgemäß ist der Bytecode-Verifier derart ausgebildet, daß beim Prüfen des Computerprogramms auch überprüft wird, ob vor dem Zugriff auf Variablen diese vorher bereits initialisiert worden sind, das heißt, mit einem Wert belegt worden sind, so daß beim Zugriff auf die jeweilige Variable ein definierter Wert gelesen werden kann und kein undefinierter Zustand erzeugt wird.

Nachfolgend wird die Routine zum Prüfen des Zugriffes auf die Variablen im Pseudo-Code dargestellt:

```
while isOK ∧ (locVar ≤ numberVars)
  then if frontier = []
            locVar := locVar + 1
            visited := ∅
            frontier := [1]
       else first := car(frontier)
            rest := cdr(frontier)
            visited := {first} ∪ visited
            if initVar(P(first),locVar)
               then frontier := rest
               else if readVar(P(first),locVar)
                       then isOk := false
                       else frontier := select (possibleNext
                            (first), visited) * rest
                    end-if
            end-if
       end-case
end-while
```

In Fig. 2 ist ein Flußdiagramm dargestellt, das die oben im Pseudo-Code dargestellte Routine zeigt.

Die einzelnen Elemente dieser Routine haben folgende Bedeutung:

isOk ist eine Boolsche Variable, die die Werte wahr und falsch annehmen kann. Sie wird vorab mit dem Wert wahr belegt und falls bei der Überprüfung des Computerprogrammes ein unzulässiger Datenverarbeitungsvorgang festgestellt wird, mit falsch belegt.

locVar ist ein Zähler, der mit ganzzahligen positiven Zahlen belegt werden kann. locVar dient zum Zählen der lokalen Variablen eines entsprechenden Feldes. locVar wird mit der Anzahl z der Parameter initialisiert, die beim Aufruf des Programmes an dieses übergeben werden. Hierbei wird davon ausgegangen, daß die Parameter aufeinanderfolgend auf die Variablen 1 bis z abgebildet werden.

numberVars ist eine Funktion, die eine ganzzahlige Zahl ausgibt, die der Länge des Variablenfeldes entspricht.

frontier ist eine Liste, die mehrere ganzzahlige Elemente beinhalten kann, wobei in dieser Liste Nummern von Befehlen enthalten sind, die noch zu prüfen sind.

visited ist eine Menge von ganzzahligen Zahlen, die die Nummern der bereits geprüften Befehle beinhaltet.

car(Liste) ist eine Funktion, die das erste Element einer Liste liefert.

cdr(Liste) ist eine Funktion, die den Rest der Liste liefert, wobei der Rest alle Elemente der Liste bis auf das erste Element umfaßt.

initVar (instr, n) ist eine Funktion, die ein wahr ergibt, falls der Befehl instr einen Wert in die lokale Variable mit der Nummer n schreibt, wodurch diese Variable initialisiert wird.

readVar (instr, n) ist eine Funktion, die wahr ergibt, falls der Befehl instr die lokale Variable mit der Nummer n liest und damit auf diese lokale Variable zugreift.

possibleNext (1) ist eine Funktion, die die Liste aller Nummern der Befehle ergibt, die bei der Ausführung des Befehls mit der Nummer 1 auf diesen folgen können. Enthält der Befehl 1 keine Sprunganweisung, so kann auf diesen Befehl lediglich ein einziger Befehl folgen, so daß possibleNext lediglich einen einzigen Wert ergibt. Enthält der Befehl mit der Nummer 1 hingegen eine bedingte Sprunganweisung, so können je nach Erfüllung der Bedingung zwei unterschiedliche Befehle auf den Befehl mit der Nummer 1 folgen, wodurch die Funktion possibleNext eine Liste mit zwei Werten ergibt.

select (l, s) ist eine Funktion, die die Elemente von der Liste l entfernt, die in der Menge s vorhanden sind.

l1 * l2 ist eine Funktion, die die Elemente der Liste l2 an die Elemente der Liste l1 anhängt.

Nachfolgend wird die Routine anhand der obigen Darstellung im Pseudo-Code und anhand von Fig. 2 näher erläutert. Die Routine wird mit dem Schritt S1 gestartet (Fig. 2). Danach wird abgefragt, ob isOk wahr ist und ob locVar kleiner als oder gleich zu numberVars ist (S2). Wenn diese Abfrage wahr ergibt, geht der Programmablauf auf den Schritt S3 über. Ergibt die Abfrage den Wert falsch, so wird auf das Ende des Programmes (S4) verzweigt, da entweder alle lokalen Variablen getestet worden sind – wenn locVar größer als numberVars ist – oder wenn der Wert von isOk falsch ist, das heißt, wenn ein unzulässiger Datenverarbeitungsvorgang festgestellt worden ist. In der oben angegebenen Routine wird diese Abfrage in Form einer if-, then-, end-if-Anweisung ausgeführt.

Im Schritt S3 wird abgefragt, ob die Liste frontier leer ist. Falls die Liste frontier leer ist, geht der Programmablauf auf den Schritt S6 über, mit dem der Zähler locVar um eins erhöht wird, die Menge visited mit der leeren Menge belegt wird

und die Liste frontier mit einer Liste, die alleine aus dem Element 1 besteht, belegt wird. Nach Ausführung des Schrittes S6 geht der Programmablauf wieder auf den Schritt S2 über, wobei dann im Schritt S3 die Abfrage den Wert falsch ergibt, weil die Liste frontier ein Element beinhaltet. Der Programmablauf geht auf den Schritt S5 über, bei dem zu der Menge visited das jeweils erste Element der Liste frontier hinzugefügt wird, da in den nachfolgenden Programmschritten der dem ersten Element der Liste frontier entsprechende Befehl überprüft wird.

In Schritt S7 wird abgefragt, ob der Befehl P(first) des Programmes P mit der Nummer first die lokale Variable mit der Nummer locVar initialisiert. Diese Abfrage wird mittels der oben erläuterten Funktion initVar ausgeführt. Ist das Ergebnis dieser Abfrage der Wert falsch, so geht der Programmablauf auf die Abfrage S8 über, mit der geprüft wird, ob der Befehl P(first) mit der Nummer first die lokale Variable mit der Nummer locVar liest, das heißt auf sie zugreift. Ergibt die Abfrage im Schritt S8, daß der Befehl P(first) mit der Nummer first auf diese lokale Variable zugreift, so wird im Schritt S9 die Variable isOk mit dem Wert falsch belegt.

Ergibt die Abfrage im Schritt S8 hingegen, daß der Befehl P(first) mit der Nummer first nicht auf diese lokale Variable zugreift, das heißt daß der Wert der Abfrage falsch ist, verzweigt der Programmablauf auf den Schritt S10, in dem die Liste frontier neu belegt wird. Die Belegung der Liste frontier erfolgt mit der Funktion possibleNext von first, die alle Programmbefehle ermittelt, die auf den Befehl mit der Nummer first folgen können. Weist dieser Befehl einen bedingten Sprung auf, so können auf diesen Befehl zwei unterschiedliche Befehle folgen, wohingegen bei einem Befehl ohne Sprung oder mit einem unbedingten Sprung lediglich ein einziger Befehl folgen kann. Das Ergebnis der Funktion possibleNext wird mit der Menge visited mittels der Funktion select verknüpft, so daß lediglich die auf den Befehl mit der Nummer first folgenden Befehle in der Liste frontier verbleiben, die noch nicht geprüft worden sind. Ferner verbleiben alle restlichen Befehle, bzw. die darauf zeigenden Nummern, in der Liste frontier. Mit dem Schritt S10 werden somit der Liste frontier alle auf den aktuell geprüften Befehl folgenden Befehle hinzugefügt, die noch nicht geprüft worden sind.

Falls die Abfrage im Schritt S7 ergibt, daß der Befehl P(first) mit der Nummer first die lokale Variable mit der Nummer locVar initialisiert, das heißt mit einem Wert belegt, ist das Ergebnis der Abfrage wahr und der Programmablauf geht auf den Schritt S11 über, mit dem aus der Liste frontier das Element first ersatzlos entfernt wird und in der lediglich die übrigen Elemente (rest) verbleiben. Von den Schritten S9, S10 und S11 geht der Programmablauf wieder zurück auf den Schritt S2.

Nachfolgend wird die Funktionsweise dieser Routine anhand des in **Fig.** 3 gezeigten Programmbeispieles näher erläutert.

Die Routine beginnt mit dem Schritt S1, wobei zu Beginn des Programms der Menge visited eine leere Menge und der Liste frontier eine leere Liste zugewiesen wird und die Variable isOk mit dem Wert wahr belegt ist. Der Zähler locVar ist auf 1 gesetzt, da ein Parameter dem Programm übergeben wird (1→C; siehe **Fig.** 3).

In dem Programmbeispiel gibt es zwei Variablen, weshalb im Schritt S2 der Inhalt von locVar, der zunächst 1 ist, kleiner als das Ergebnis numberVars ist, das gleich 2 ist, weshalb die Abfrage locVar ≤ numberVars den Wert wahr ergibt und da auch der Wert der Variable isOk wahr ist, geht der Programmablauf auf den Schritt S3 über. Da die Liste frontier leer ist, geht der Programmablauf auf den Schritt S6 über, in dem der Wert des Zählers locVar um eins erhöht wird und somit 2 ist, der Satz visited wieder mit der leeren Menge belegt wird und die Liste frontier mit der Liste belegt wird, die als einziges Element die Zahl 1 aufweist.

Eine erneute Abfrage im Schritt S2 ergibt wieder den Wert wahr, da isOk unverändert wahr ist und der Wert des Zählers locVar gleich der Anzahl der Variablen ist. Die nachfolgende Abfrage im Schritt S3 ergibt den Wert falsch, da nun die Liste frontier das Element 1 beinhaltet. Der Programmablauf geht somit auf den Schritt S5 über. Im Schritt S5 wird das erste Element von der Liste frontier, im vorliegenden Fall das einzige Element, nämlich das Element 1, der Menge visited hinzugefügt.

Im darauffolgenden Schritt S7 wird mittels der Funktion initVar geprüft, ob der Programmbefehl mit der Nummer 1 (goto 5) die Variable 2 initialisiert.

Die Abfrage im Schritt S7 ergibt für den Befehl 1 und die Variable 2 den Wert falsch, da die Variable 2 nicht mit dem Befehl 1 initialisiert wird. Der Programmablauf geht deshalb auf die Abfrage S8 über, mit der geprüft wird, ob der Befehl 1 (goto 5) die Variable 2 liest. Da dies nicht der Fall ist, ist der Wert der Abfrage S8 falsch, womit der Programmablauf auf den Schritt S10 übergeht. Im Schritt S10 werden der Liste frontier die auf den Befehl 1 folgenden Befehle im Programmablauf zugewiesen, die noch nicht geprüft worden sind. Beim Programmbeispiel folgt auf den Befehl 1 der Befehl 5 (aload 2), da der Befehl 1 (goto 5) einen Sprung auf den Befehl 5 ausführt.

Der Programmablauf geht wiederum über die Schritte S2, S3 auf den Schritt S5 über, mit dem der Menge visited das Element 5 für den Befehl Nummer 5 hinzugefügt wird. Die Abfrage im Schritt S7 ergibt, daß der Befehl 5 (aload 2) nicht die Variable 2 initialisiert. Der Wert der Abfrage S7 ist somit falsch und der Programmablauf verzweigt auf den Schritt S8. Die Abfrage im Schritt S8 ergibt, daß der Befehl 5 die Variable 2 liest (aload 2), wodurch das Ergebnis der Abfrage S8 wahr ist und der Programmablauf auf den Schritt S9 verzweigt.

Dieses Ergebnis der Abfrage im Schritt S8 bedeutet, daß die Variable 2 mit dem Programmbefehl 5 gelesen worden ist, ohne daß sie vorher initialisiert worden ist. Hiermit würde beim Ausführen des Beispielprogrammes ein undefinierter Programmzustand entstehen. Dies ist in **Fig.** 3 mit dem Fragezeichen im Stack dargestellt. Bei der Prüfung des Beispielprogrammes wird deshalb im Schritt S9 die Variable isOk mit dem Wert falsch belegt. Der Programmablauf geht nun auf den Schritt S2 über, dessen Abfrage den Wert falsch ergibt, da die Variable isOk mit dem Wert falsch belegt ist. Der Programmablauf verzweigt somit vom Schritt S2 auf den Schritt S4, womit die Routine beendet ist.

In **Fig.** 4 ist schematisch der Programmablauf des Programmbeispieles aus **Fig.** 3 gezeigt, wobei jeder Befehl durch einen Punkt mit der dazugehörigen Nummer dargestellt ist und von den einzelnen Befehlen jeweils Pfeile zu dem im Programmablauf folgenden Befehl zeigen. Mit der oben beschriebenen Routine wird die Prüfung der Variablen 1 übersprungen, da sie durch die Parametereingabe vorab initialisiert ist und deshalb locVar auf 1 gesetzt ist. Die Befehle 1 und 5 werden in der Reihenfolge des Programmablaufes abgearbeitet, um eine Zugriffsverletzung auf die Variable 2 zu prüfen. Da beim Befehl 5 eine derartige Zugriffsverletzung auftritt, wird diese festgestellt und die weitere Prüfung abgebrochen.

Fig. 5 zeigt einen Programmablauf eines weiteren Programmbeispieles, das nicht im Detail beschrieben wird. Dieses

Programmbeispiel weist wiederum zwei Variablen auf, wobei beim Befehl 3 ein bedingter Sprung auf den Befehl 7 vorgesehen ist. Die Befehle 1, 2, 3, 4 und 8 initialisieren weder eine der beiden Variablen, noch greifen sie darauf zu. Der Programmablauf verzweigt nach dem Befehl 3 auf den Befehl 4 bzw. auf den Befehl 7 und die jeweiligen danach kommenden Befehle. Mit dem Befehl 5 (astore 1) wird die Variable 1 initialisiert und mit dem Befehl 7 (astore 2) wird die Variable 2 initialisiert. Bei der Prüfung dieses Programmbeispieles durch die oben angegebene Routine wird das Programmbeispiel auf einen unzulässigen Zugriff auf die Variable 1 geprüft. Hierbei werden die einzelnen Befehle in der Reihenfolge 1, 2, 3, 4, 5, 7, 8, 9 geprüft. Nach der Abarbeitung des Befehles 3 werden in die Liste frontier die Befehle 4 und 7 aufgenommen, da beide dem Befehl 3 folgen können. Zunächst werden die Befehle 4 und 5 abgearbeitet, wobei dieser Zweig beim Befehl 5 abgebrochen wird, da hier die Variable 1 initialisiert wird. Danach werden die Befehle 7, 8 und 9 abgearbeitet.

Bei der Prüfung einer Zugriffsverletzung bzgl. der Variable 2 werden in entsprechender Weise die Befehle in der Reihenfolge 1, 2, 3, 4, 5, 6 und 7 abgearbeitet, wobei wiederum die Verzweigung nach dem Befehl 3 berücksichtigt wird und die Bearbeitung des mit dem Befehl 7 beginnenden Zweiges am Befehl 7 abgebrochen wird, da hier die Variable 2 initialisiert wird.

Mit der oben beschriebenen Routine werden somit die Programmbefehle eines Programmes in der Reihenfolge des Programmablaufes geprüft, wobei bei einer Verzweigung des Programmablaufes die einzelnen Zweige des Programmablaufes separat untersucht werden, bis entweder die entsprechende Variable initialisiert oder eine Zugriffsverletzung festgestellt wird.

Ein durch die erfindungsgemäße Routine ergänzter Bytecode-Verifier ermöglicht mit minimalem Speicherplatzbedarf eine vollständige Prüfung eines auf einem Computersystem geladenen Computerprogramms. Es wird lediglich zusätzlicher Speicherplatz für die Abspeicherung der Routine, der Menge visited und der Liste frontier benötigt. Die Menge visited ist aber höchstens so groß wie das zu prüfende Programm. Das gleiche gilt für die Liste frontier. Der Speicherplatzbedarf ist somit wesentlich geringer als bei einem bekannten Bytecode-Verifier, bei dem das virtuelle Feld für die lokalen Variablen für jede Sprungadresse kopiert wird.

Der mit der erfindungsgemäßen Routine versehene Bytecode-Verifier kann in einem Computersystem installiert auf einem Datenträger gespeichert oder über ein Datennetz auf ein Computersystem geladen werden. Ein derartiger Bytecode-Verifier wird zum Prüfen des Programmcodes von Computerprogrammen, insbesondere von welchen, die in einer plattformunabhängigen Sprache, wie z. B. OAK oder Java geschrieben sind, verwendet. Dieser Programmcode wird auch als Bytecode bezeichnet.

Die Erfindung kann folgendermaßen zusammengefaßt werden.

Die Erfindung betrifft ein Verfahren zum Betreiben eines Computersystems, so wie einen Bytecode-Verifier und ein Computersystem.

Mit dem erfindungsgemäßen Verfahren wird geprüft, ob ein auf ein Computersystem geladenes Computerprogramm einen unzulässigen Zugriff auf eine Variable ausübt, das heißt, ob die Variable bevor sie gelesen wird, initialisiert ist. Diese Prüfung erfolgt vor der Ausführung des Programmes, so daß beim Ausführen des Programmes eine derartige Prüfung nicht mehr durchgeführt werden muß.

Das erfindungsgemäße Verfahren benötigt bei der Prüfung des Programmes wenig Speicherplatz und führt dennoch eine vollständige Prüfung durch. Da die Prüfung vor der Ausführung des Programmes erfolgt, wird die Programmausführung selbst erheblich beschleunigt, da hierbei keine weitere Prüfung notwendig ist.

Patentansprüche

1. Verfahren zum Betreiben eines Computersystems, das folgende Schritte umfaßt:
   a) Speichern eines Computerprogramms in einem Speicher (3, 4) des Computersystems (1), wobei das Computerprogramm eine Folge von Bytecodes umfaßt, die Programmbefehle darstellen, wobei die Programmbefehle in der Reihenfolge eines vorbestimmten Programmablaufes bei der Ausführung des Computerprogrammes abgearbeitet werden und auf Variablen zugreifen können,
   b) Prüfen des in das Computersystem (1) geladenen Computerprogrammes vor dem Ausführen des Computerprogramms, ob die darin enthaltenen Programmbefehle beim Ausführen einen unzulässigen Datenverarbeitungsvorgang erzeugen würden, und
   c) falls das Prüfen des Computerprogramms keinen unzulässigen Datenverarbeitungsvorgang ergeben hat, wird die Ausführung des Programms erlaubt, ansonsten wird eine entsprechende Fehlermeldung ausgegeben,
   dadurch gekennzeichnet, daß beim Prüfen des Computerprogramms die einzelnen Programmbefehle in der Reihenfolge des Programmablaufes überprüft werden, ob sie bei Ihrer Ausführung auf eine Variable zugreifen, bevor diese initialisiert worden ist.
2. Verfahren zum Betreiben eines Computersystems nach Anspruch 1, dadurch gekennzeichnet, daß beim Prüfen des Computerprogramms ein virtueller Stack überprüft wird, ob durch einen unzulässigen Datenverarbeitungsvorgang ein Overflow oder ein Underflow des virtuellen Stacks erzeugt wird.
3. Verfahren zum Betreiben eines Computersystems nach Anspruch 1 oder 2, dadurch gekennzeichnet, daß ein virtuelles Feld lokaler Variablen erzeugt wird, in dem anstelle der Daten und Konstanten, die bei der Ausführung des Computerprogrammes in ein korrespondierendes tatsächliches Feld lokaler Variablen eingetragen werden, Indikatoren gespeichert werden, die den jeweiligen Datentyp darstellen.
4. Verfahren zum Betreiben eines Computersystems nach Anspruch 3, dadurch gekennzeichnet, daß beim Prüfen des Computerprogramms für jede lokale Variable des Feldes aufeinanderfolgend geprüft wird, ob sie vor den Zugriffen initialisiert worden ist.
5. Verfahren zum Betreiben eines Computersystems nach einem der Ansprüche 1 bis 4, dadurch gekennzeichnet, daß beim Prüfen des Computerprogramms die Programmbefehle in der Reihenfolge des Programmablaufes abgefragt werden, ob durch den jeweiligen Programmbefehl eine bestimmte Variable initialisiert wird (S7), und

falls diese Abfrage ergibt, daß die Variable durch diesen Programmbefehl nicht initialisiert wird, abgefragt wird, ob durch diesen Programmbefehl auf diese Variable zugegriffen wird (S8), und
falls diese Abfrage ergibt, daß auf diese Variable zugegriffen wird, eine Fehlermeldung ausgegeben und die Ausführung des Computerprogramms nicht erlaubt wird (S9).

6. Verfahren zum Betreiben eines Computersystems nach Anspruch 5, dadurch gekennzeichnet, daß wenn sich bei der Abfrage nach dem Zugriff auf die Variable ergibt, daß auf die Variable nicht zugegriffen wird, alle Programmbefehle ermittelt werden, die auf den soeben geprüften Programmbefehl im Programmablauf folgen können, und ermittelt wird, ob einer oder mehrere dieser Programmbefehle schon geprüft worden sind (S10), und falls sich hierbei einer oder mehrere ungeprüfte Programmbefehle ergeben, werden diese gemäß den Verfahrensschritten im Anspruch 5 geprüft.

7. Verfahren zum Betreiben eines Computersystems nach Anspruch 6, dadurch gekennzeichnet, daß wenn sich kein ungeprüfter Programmbefehl ergibt, das Computerprogramm nach der nächsten Variablen (S11) geprüft wird und falls alle Variablen geprüft worden sind, wird der Prüfvorgang beendet.

8. Verfahren zum Betreiben eines Computersystems nach einem der Ansprüche 5 bis 7, dadurch gekennzeichnet, daß wenn sich bei der Abfrage, ob durch den jeweiligen Programmbefehl eine bestimmte Variable initialisiert wird, ergibt, daß die Variable durch den Programmbefehl initialisiert wird, abgefragt wird, ob weitere Programmbefehle, die nicht auf den soeben geprüften Programmbefehl folgen, auf diese Variable geprüft werden müssen, und ergeben sich derartige weitere Programmbefehle, so werden diese gemäß den Verfahrensschritten im Anspruch 5 geprüft, ansonsten wird das Computerprogramm nach der nächsten Variablen geprüft und falls alle Variablen geprüft worden sind, wird der Prüfvorgang beendet.

9. Bytecode-Verifier, der ein auf ein Computersystem geladenes Programm auf unzulässige Datenverarbeitungsvorgänge vor seiner Ausführung prüft, dadurch gekennzeichnet, daß der Bytecode-Verifier derart ausgebildet ist, daß die Programmbefehle des Computerprogramms in der Reihenfolge des Programmablaufes überprüft werden, ob sie bei ihrer Ausführung auf eine Variable zugreifen, bevor diese initialisiert wird.

10. Bytecode-Verifier nach Anspruch 9, dadurch gekennzeichnet, daß der Bytecode-Verifier zur Ausführung des Verfahrens nach einem der Ansprüche 1 bis 8 ausgebildet ist.

11. Bytecode-Verifier nach Anspruch 9 oder 10, dadurch gekennzeichnet, daß er auf einem Datenträger gespeichert ist.

12. Computersystem mit
einem Speicher zum Speichern eines Computerprogramms, wobei das Computerprogramm eine Folge von Bytecodes umfaßt, die Programmbefehle darstellen, wobei einige Programmbefehle auf Variablen zugreifen können,
einer Datenverarbeitungseinheit zum Ausführen von im Speicher gespeicherten Programmen, wobei
ein Bytecode-Verifier im Speicher gespeichert ist, der ein in den Speicher des Computersystems geladenes Programm auf unzulässige Datenverarbeitungsvorgänge vor seiner Ausführung prüft,
dadurch gekennzeichnet, daß der Bytecode-Verifier derart ausgebildet ist, daß das Computerprogramm überprüft wird, ob bei seiner Ausführung vor den möglichen Zugriffen auf Variablen diese jeweils initialisiert worden sind.

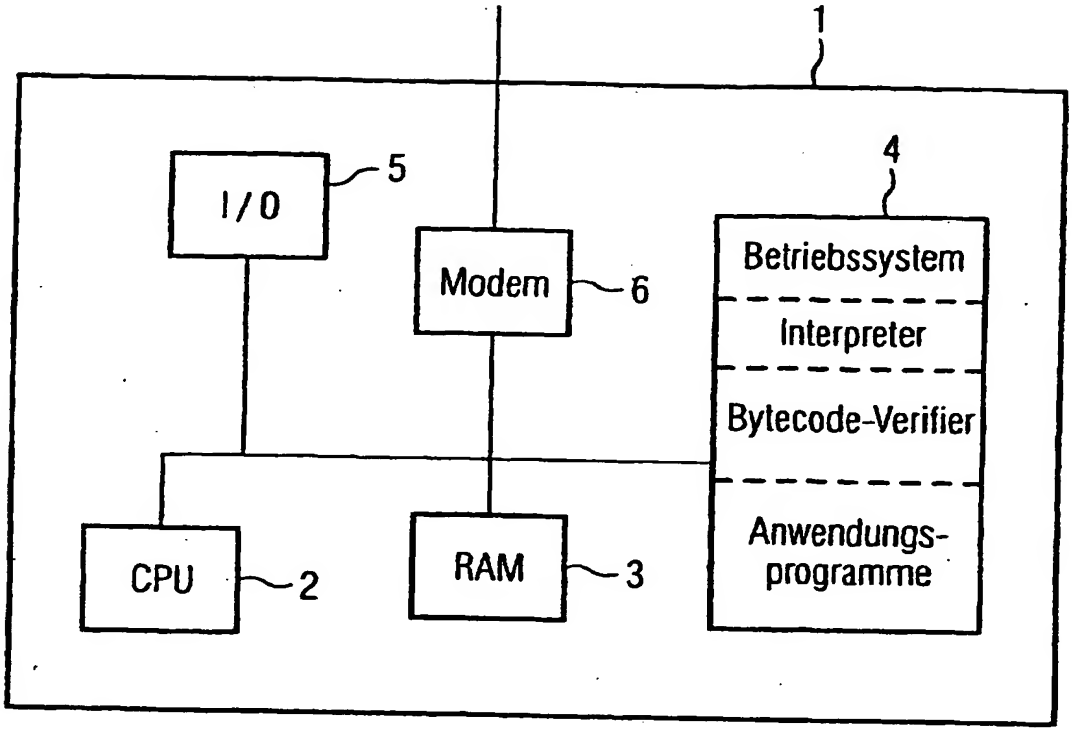
13. Computersystem nach Anspruch 12, dadurch gekennzeichnet, daß der Bytecode-Verifier gemäß den Ansprüchen 9 bis 11 ausgebildet ist.

Hierzu 4 Seite(n) Zeichnungen

- Leerseite -

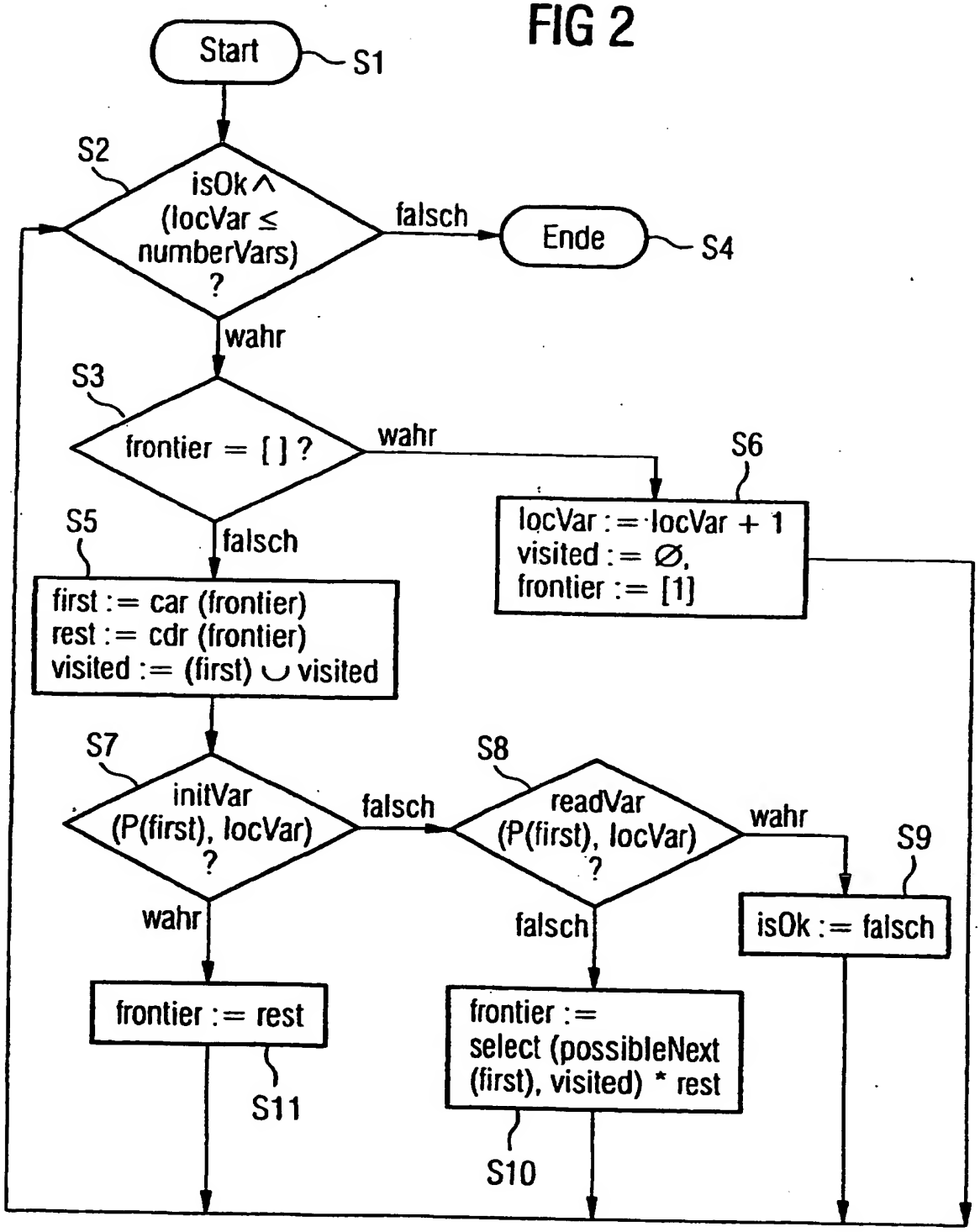
Nummer: DE 199 26 467 C1
Int. Cl.[7]: G 06 F 11/36
Veröffentlichungstag: 11. Januar 2001

FIG 1



Nummer: DE 199 26 467 C1
Int. Cl.[7]: G 06 F 11/36
Veröffentlichungstag: 11. Januar 2001

FIG 2



Nummer: DE 199 26 467 C1
Int. Cl.[7]: G 06 F 11/36
Veröffentlichungstag: 11. Januar 2001

FIG 3

Befehl	virtueller Stack	lokale Variablen
	[ ]	{1→ C, 2 → ?}
1 goto 5		
	[ ]	{ }
2 aload 1		
	[ ]	{ }
3 astore 2		
	[ ]	{ }
4 goto 9		
	[ ]	{1→ C, 2 → ?}
5 aload 2		
	[?]	{1→ C, 2 → ?}
6 getfield x		
	[ ]	{ }
7 pop		
	[ ]	{ }
8 goto 2		
	[ ]	{ }
9 return		

Nummer: DE 199 26 467 C1
Int. Cl.[7]: G 06 F 11/36
Veröffentlichungstag: 11. Januar 2001

FIG 4

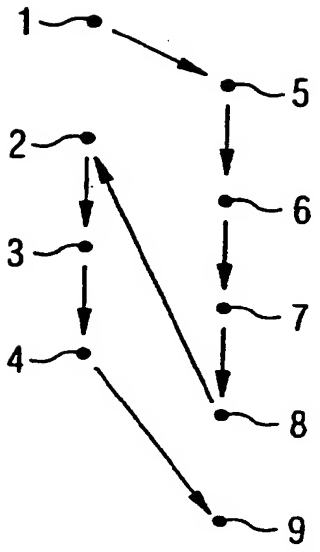


FIG 5